Knowledge Systems Laboratory
Report No. KSL 93-35

April 1993

IN-61-CR
C17-0
182779
9P

Architecture-driven Reuse of Code in KASE

by

Sanjay Bhansali

KNOWLEDGE SYSTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, California 94305

Work supported in part by NASA grant NCC 2-749-1

Architecture-driven Reuse of Code in KASE

Sanjay Bhansali
Knowledge Systems Laboratory
Computer Science Department, Stanford University
701 Welch Road, Building C
Palo Alto, CA 94304
bhansali@ksl.stanford.edu

Abstract

In order to support the synthesis of large, complex software systems, we need to focus on issues pertaining to the architectural design of a system in addition to algorithm and data structure design. In this paper, we present an approach that is based on abstracting the architectural design of a set of problems in the form of a *generic architecture*, and providing tools that can be used to instantiate the generic architecture for specific problem instances. Such an approach also facilitates reuse of code between different systems belonging to the same problem class. We describe an application of our approach on a realistic problem, present the results of the exercise, and discuss how our approach compares to other work in this area.

1. Introduction

Until recently most research in providing automated support for software synthesis has focused on issues concerned with the synthesis of algorithms and data structures (e.g. (Barstow, 1979; Smith, 1990)) It is now being realized that in order to support the synthesis of large, complex systems, we also need to focus on design problems that go beyond the level of algorithms and data structures (Bhansali & Nii, 1992b; Graves, 1991; Lubars, 1991; Mettala, 1990; Shaw, 1990). This has been called the *architectural* level of design. In the KASE (Knowledge Assisted Software Engineering) project we have developed an approach for providing automated support to a software designer in designing software systems at the architectural level. In brief, our approach can be described as follows: (1) abstract a set of different problem instances as a *problem class* (2) abstract the design of software systems for the various problem instances of a problem class as a *generic architecture*(3) corresponding to each generic architecture/problem class pair create a body of rules (called *customization knowledge*) that describes how the generic architecture can be *customized* for specific instances of the problem class, and (4) build a set of generic tools that can assist a software designer in the customization process.

We have described and demonstrated the feasibility of this approach in the design of two different systems using a common blackboard-based generic architecture for tracking(Bhansali & Nii, 1992a; Bhansali & Nii, 1992b). Our experience with the tracking domain showed that the KASE approach is a promising one for capturing design knowledge and reusing it to design new systems more efficiently and with fewer errors due to omission. The output of the customization process is a specification of the various components of the architecture which has to be transformed to code.

In this paper we describe an exercise in applying the KASE approach to another application with an executable system being the final output. We were motivated in this exercise by both academic and pragmatic reasons. Academically, we were interested in investigating (1) how our approach generalizes to a different application domain, and (2) how reuse at the architecture level enables reuse of code leading to efficient synthesis of solutions for new problem instances. This application differed from the previous one because we already had available a considerable amount of code for several problem instances and we were interested in reusing that code.

The exercise was also of pragmatic interest to us because the application that we chose is a subsystem of KASE itself. Our objective was to use KASE to help us maintain and modify that subsystem more efficiently. KASE contains tools that provide diagrammatic representation of a software system from different perspectives. Figure 1 shows examples of the output generated by some of these diagramming tools. Frequently we need to add new kinds of diagramming capabilities which may be specific to a particular domain (e.g. in a blackboard architecture, it is useful to be able to depict the interaction between a class of modules, called *knowledge sources*, and events that trigger them, in the form of a diagram). Or, a designer may want to create a diagram that shows a new perspective of an architecture (e.g. a diagram that shows both the control flow and data flow between processes).

Due to historical reasons the various diagramming tools were written by different members of the KASE project (mostly graduate students). These tools differed considerably in the implementation details (e.g. the layout algorithm used) although they all shared a common data structure for representing the architectural components. In creating new kinds of diagrams we noticed that almost all the functionality that was needed was already available (a lot of the functionality was duplicated!) in the existing code. However, considerable effort was being expended in extracting the right pieces of code, modifying them, and putting them together to create a new tool. Moreover we had to rely solely on one programmer who knew the implementation details of the various pieces of software.

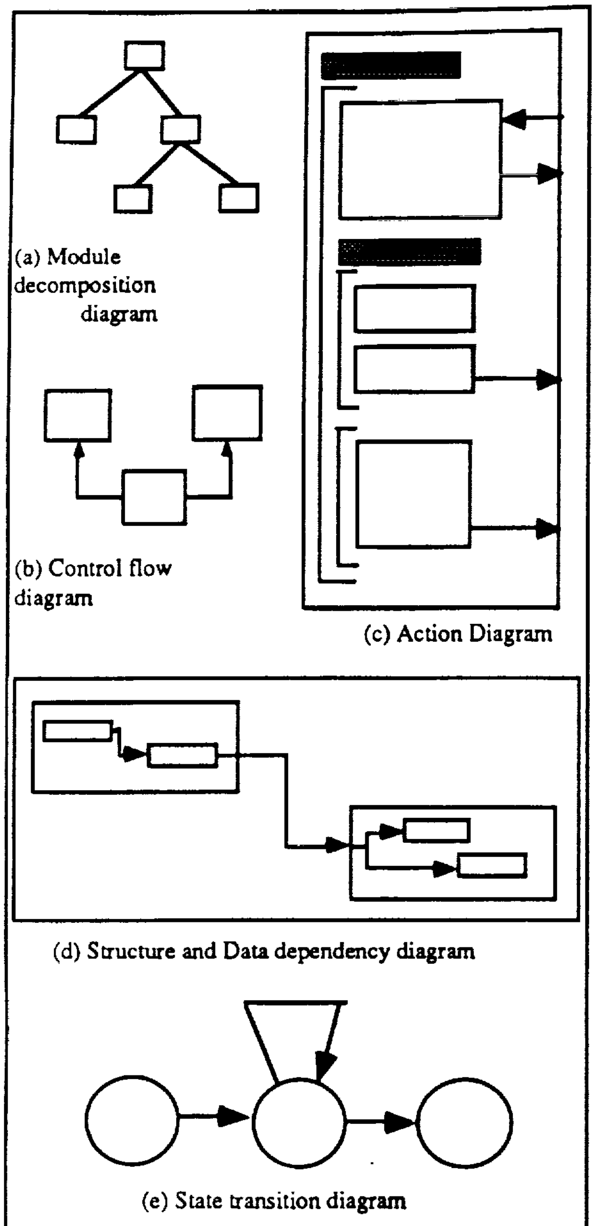
(a) Module decomposition diagram
(b) Control flow diagram
(c) Action Diagram
(d) Structure and Data dependency diagram
(e) State transition diagram

Figure 1. Examples of some diagrams generated by the graphical interface in KASE.

This is, of course, the classical maintenance problem in software engineering. Our goal was to explicitly represent the knowledge about the design of these systems in KASE and use that to maintain the code in future.

1. 1 Overview of KASE

Figure 2 illustrates the major building blocks as well as the main steps in the design process in KASE. The shadowed boxes represent knowledge components that are part of KASE. A designer initiates the design process by first selecting a generic architecture from a library based on the problem class for his particular problem and the desired solution features. Associated with a problem-class is a problem-class model which consists of generic terms pertaining to the problem class. The designer has to specify his particular problem by instantiating an abstract problem description using the generic terms from the problem-class model. The design process consists of an interactive session in which the customization knowledge associated with the generic architecture/problem class is used to refine the generic architecture based on requirements of the problem instance. Typically, during the customization process KASE provides alternative ways of customizing architectural parameters, offers default suggestions and rationales for the suggestions, maintains dependencies between various customization actions, while the designer is responsible for choosing appropriate values for the parameters based on his requirements. Finally, KASE has a constraint checker that may be used to check for the consistency of the design. A detailed description of the various components and the design environment in KASE is given elsewhere. Here, we will only describe those aspects of KASE that are related to our application problem.

2. Abstracting the Problem Description: Problem Class Model

A problem description is obtained by abstracting out the common features of a set of problems. This results in the creation of a concise, general problem specification which can be extended to create specifications for particular problems.

Analyzing the different diagrams in Figure 1, we observe that all of them essentially layout a set of objects, with the topological arrangements of the objects and the connecting lines symbolizing certain relationships between them. Also, except for the action diagram all the layouts show at most two binary relationship between the objects. The action diagram is designed to show the sequence of control flow within a procedure. It contains layout constructs to show control constructs like sequencing, conditional branches, loops, etc. in addition to the input-output dataflow and the nesting of procedures. We decided to ignore the action diagram from our problem class which resulted in a concise abstract specification for all the remaining diagramming problems. An abstract problem class is represented in terms of the inputs and outputs of the problems, a set of parameters that need to be instantiated for specific problem instances and a set of constraints that specify additional properties of the parameter values. The problem class for our diagramming problems is represented as follows:

Problem Class: Generic-Diagramming-Problem (GDP)

Inputs:- x :τ_1 *; input is any object of type* τ_1

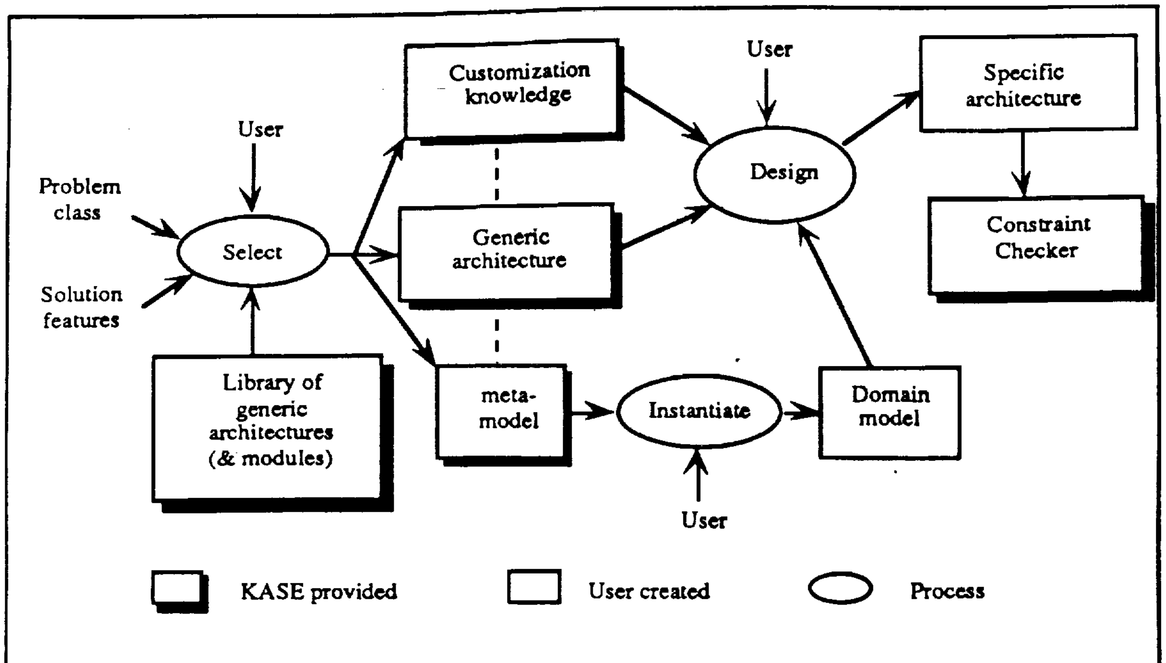

Figure 2. Overview of software design in KASE

Problem parameters:-

1) τ_1 : type *; type of the input object*

2) τ_o :type *;type of objects to be shown in the layout*

3) O : (τ_1 -> set(τ_o)) *; a lambda expression that takes as input an object of type τ_1 and returns a set of objects of type τ_o*

4) R1: set(τ_o) -> set([τ_o ,τ_o]) *; a lambda expression that takes as input a set of objects of type τ_o and returns a set of ordered pairs [τ_o ,τ_o]*

5) R2: set(τ_o) -> set([τ_o ,τ_o]) *; another lambda expression to compute a second set of ordered pairs defining another relation on the objects τ_o .*

Parameter constraints:- None

Output:- *w* :KEE-picture-window

Postcondition:- contains-layout(w,layout(O,R1(O(x)), R2(O(x))))

Annotation: "Given <O, R1, R2> where O is a set of objects, R1 and R2 are binary relations on objects in O, draw a layout diagram that shows the objects in O and the relations R1 and R2 between the objects."

The abstract problem descriptions are used as indices to a library of generic architecture-level designs for problem classes. The English annotations attached to problem classes help a user in determining the appropriate problem class for his problem. In general, there may be more than one architecture that can be used to solve a problem. Each architecture is characterized by certain properties determining the nature of the solution (for example, a tracking problem may be solved by a statistical analysis of the data or a symbolic interpretation - the architectures for the two solutions would be quite different). These properties are called solution features of the architecture and are used to select among different architectures for a problem. For the generic-diagramming-problem we created just one architectural solution and so the solution features are not needed.

Once a problem class is selected by a user she has to specify her problem as an instance of the problem class. Thus, for the GDP example, the user has to specify the type of an input object (these are the objects on which a user can click at run-time to request the specified diagram), methods to determine the set of objects to be shown on the layout, methods to compute the relations R1 and R2 between the objects in the set, and any additional properties of the parameters that may help in determining an appropriate diagram for the problem. In order to assist a user in specifying individual problems belonging to a problem class, a domain theory is built which contains a collection of generic concepts which can be used to formulate problem instances. The collection of generic concepts may be viewed as a model for an abstract domain formed by abstracting a set of different problem domains, and is, therefore, termed a *problem-class model*.

The development of a problem-class model begins by defining the terms used in the abstract problem description. The terms used in the GDP description are object, binary-relation, and layout. An object is any element in the universe and a binary-relation is defined mathematically as an ordered pair of objects. The definitions of terms typically introduces new terms which have to be further defined.

Definition. A layout is a diagram consisting of nodes and connections between the nodes.
Definition. A node is a primitive shape, e.g. a point, a circle, or a rectangle.
Definition. A connection is either an edge-connection between two nodes or a topological-connection between them.

Examples of topological-connection are *nesting* and *tiering* (Figure 3). Instances of edge connections include *undirected lines, directed-lines,* and *directed arcs.* The various terms identified in this manner are organized in a class hierarchy. Figure 4 shows part of this class hierarchy for the GDP domain. Also associated with each object in the problem-class model is a set of operations that are permissible on that object. For example, an object may be created, deleted, or renamed. A node may be moved to another location or reshaped. The operations and attributes are typically defined in terms of more general objects and are inherited by objects that are subclasses of the general object.

The terms in the problem-class model also provide the vocabulary for formulating the rules that constitutes the customization knowledge for the generic architecture. Thus, the problem-class model serves to bridge the gap between a problem specification and the generic architecture. Section 4 gives examples of customization rules for the GDP domain.

3. Abstracting the solution : Generic Architecture

Just as a problem class is an abstraction of a set of problem instances, a generic architecture is an abstraction of the solutions for a set of problems. It is obtained by abstracting the common features from the solutions of problems that are instances of a problem class. Figure 5

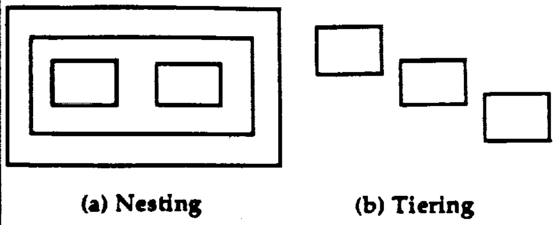

Figure 3. Examples of topological connections. (a) Nesting: A box A nested inside another box B represents a relation between A and B. (b) Tiering: A box to the right and below a box B represents a relation between A and B.

shows a generic architecture, in terms of the top-level modules obtained by abstracting the common features from the solutions of problems that are instances of a problem class. Figure 5 shows a generic architecture, in terms of the top-level modules, for the GDP problem class. The generic architecture consists of 7 main modules. The m-Diagram-Manager is the top-level module which keeps track of all active windows and the contents of that window, and interacts with the m-Editor Subsystem (not shown in the figure) of KASE as well as with the end-user. The m-Diagram-Redisplayer module is responsible for detecting and updating the contents of the various windows based on the inputs received from the m-Editor Subsystem. Thus, if a user modifies a component of a system, the m-Diagram-Redisplayer will automatically gray over and then re-display all those windows which contain that component. The m-Manipulate-Windows module contains procedures that manipulate an entire window. For example, operations that move, reshape, or close a window are provided by this module. The m-Manipulate-Diagrams module contains code that is responsible for manipulating the individual objects displayed in a window. The functionality of this module is divided into two modules: (1) m-Create-Diagram, which is

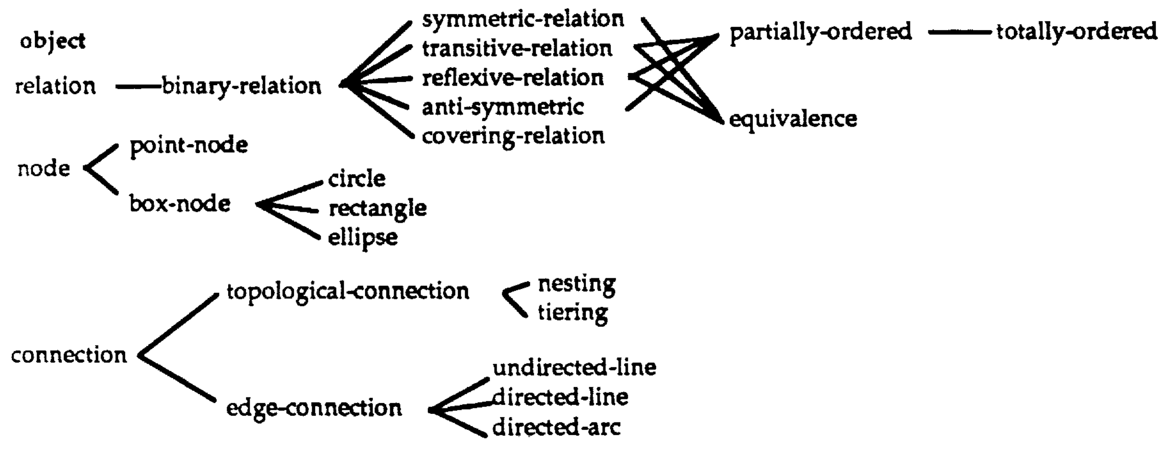

Figure 4. Generic concepts for the GDP domain.

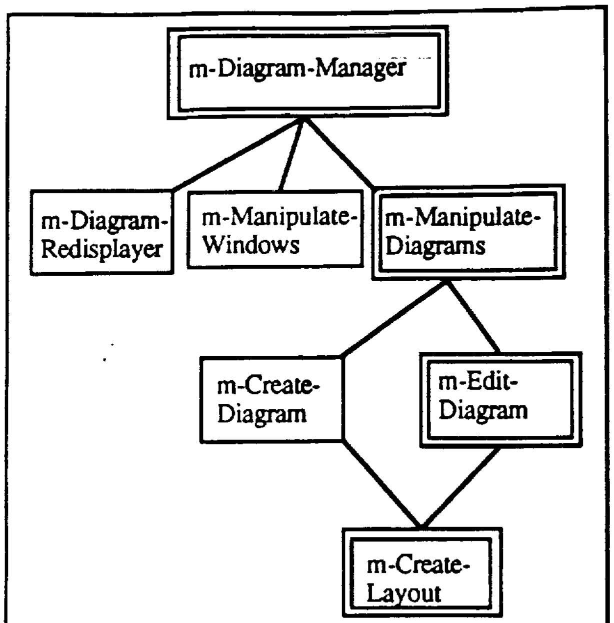

Figure 5. Generic architecture for the GDP problem class. The boxes with double outlines indicate customizable modules. In KASE customizable modules are shown highlighted.

responsible for determining the initial layout (m-Create-Layout) and displaying it in a window; and (2) m-Edit-Diagram, which displays the menu of available edit actions that can be performed on the individual objects.

In KASE, a module is represented as an object with a set of attributes. Figure 7 shows the minimal set of attributes for each module. Attributes that are preceded by an * are derived attributes whose values are computed from the primitive attributes (e.g., the input to a module is simply the set of data-types that form arguments to procedures provided by the module and the results of procedures required by the module). A module interface is defined in terms of the resources (procedures and data) that it provides to other modules, and the resources it requires from other modules. The other attributes constrain the way a system is structured and the way modules communicate with each other. For example, a module may only use resources provided by its submodules or a module that it has access to.

MODULE

supermodule	*module that contains this module*
submodules	*modules contained within this module*
provides	*resources provided by this module*
requires	*resources required by this module*
has-locally	*local resources*
has-access-to	*modules which can provide resources to this module*
*inputs	*data flow into the module*
*outputs	*data flow out of the module*
*calls	*modules called by procedures within this module*
*called-by	*modules that call procedures provided by this module*

Figure 6. Minimal internal representation of a module.

A generic module contains an abstract data type whose operations are specified (similar to the notion of generic packages in ADA), or an abstract procedure specified in terms of the inputs, outputs, preconditions, and postconditions of the procedure, or a program template that needs to be refined. For example the module m-create-layout contains the specification of a procedure: p-determine-layout(v, e, w) that takes a set of vertices *v*, edges (pairs of vertices) *e*, and a window *w*, and displays the graph of vertices and edges in the window. Different layout algorithms that take a set of vertices and edges and create a layout on a window are collected in a library indexed by a customization rule. During customization this rule is used to retrieve this set of different algorithms and depending upon her requirements the user selects one of them. It is also possible to store formal descriptions of library routines using pre-conditions and postconditions and have a theorem-prover that automatically classifies and retrieves all routines whose pre- and post-conditions match those of the generic module. Such an approach has been used in other systems (Mark,Tyler,McGuire, & Schlossberg, 1992).

Customizing an architecture refers to the process of customizing each of its generic modules. KASE provides active support to a user in customizing an architecture by providing a list of customization actions that need to be performed for each module, suggesting ways for doing the customization, and providing rationales for its suggestions. The knowledge for providing this support is represented as *customization knowledge* for each generic architecture.

4. Customization Knowledge

The customization knowledge is represented as a tuple (P, C, A, S, R, D) attached with each customizable parameter of the architecture, where:

- P is the name of the customization parameter
- C is an annotation in English explaining the role of the parameter
- A is a pointer to a set of rules that may be used in computing various alternatives for instantiating the parameter
- S is a rule that is used to suggest a default value for the parameter.
- R (optional) is a rationale that may be entered by a designer when a particular parameter value is chosen, and

• D is a list of other customization parameters or problem requirements upon which the value of this parameter depends.

An example of a customization tuple is the following:

P - layout-algorithm
C - "The algorithm used to determine the layout of nodes and their connections."
A - Rules for layout alternatives (see below)
S - Rule for determining default layout algorithm (see below)
D - Constraints(R1), Constraints(R2)

The type of layout algorithm to be used depends on the properties of the relations R1 and R2 in the problem specification. Examples of some rules that are used to suggest alternatives for a layout algorithm are the following:

Rule: If R1 is neither reflexive nor irreflexive, and R2 is empty then the tree algorithm or the nested algorithm cannot be used (because those algorithms do not permit self-loop on nodes).

Rule: If R1 is not a partial order or the length of the longest chain is greater than 3, then the nested algorithm cannot be used.

Rule: If R1 is a one-to-many relation and R2 is irreflexive or reflexive, then use a combination of nested and nearest-neighbor algorithm. (R1 is depicted by nesting and R2 by using the nearest neighbor algorithm).

These rules are used in conjunction with the specification of a problem instance to determine the set of possible layout algorithms for the problem instance. The set of all layout algorithms that have not been ruled out by the above rules then becomes the set of available alternatives. The user can ask KASE to provide default suggestions on the best algorithm to use. The default rule (S) is used to determine the best layout algorithm based on the problem requirements:

Rule: If there is only one alternative available, choose that; else if both nearest neighbor and tiered are available then if the number of edges are likely to be large use tiered, else nearest neighbor; else ...

It is possible that a problem specification is not completely specified for KASE to be able to determine the best layout. In that case, KASE prompts the user for the missing requirements. Thus, in the above example, KASE would prompt the user for an estimate about the number of edges expected for a particular relation. This information is recorded as part of the problem specification and together with the customization rule serves as a documentation for the design.

Once a suggested value for a customization parameter has been given, a designer has 3 options. 1) Choose the suggested value for the parameter. KASE will instantiate the customization parameter and mark it as being customized. 2) Ignore the suggested value and use one of the other alternatives presented earlier. KASE then prompts the user to provide a rationale for that choice. The rationale (a text string) is entered and stored along with the customization parameter (R). This rationale serves as a design documentation which can be used by future maintainers to understand why a particular design choice was made. It can also be used to refine the set of customization rules. 3) Introduce a new kind of layout algorithm. In such a situation, KASE will inform the user that the user has to provide the code for such an algorithm. The code would have to conform to the specifications of a layout algorithm in terms of the inputs, outputs, preconditions, and postconditions.

The last field of the customization tuple is used by KASE to determine what parts of an architectural design need to be reconsidered when there are changes in requirements or when a designer retracts her decision on one of the customization parameters. In the above example, the customization parameter depends on the properties of the relations R1 and R2 of a particular problem instance. But in general, they can also be other customization parameters. For example, one of the customization parameters in the m-create-diagram module is the shape of the nodes to be used. The value of this parameter depends on the layout algorithm used (e.g. for a nested layout only rectangles can be used to represent nodes).

5. Constraint Checking

As remarked earlier, it is possible for a designer to ignore the customization knowledge and the suggestions offered by KASE and manually customize the architecture. This may introduce errors in the design. Most design tools contain domain-independent constraints to check for the syntactic consistency of a design (e.g., each module has at least one input and output, a named procedure is not provided by two different modules). KASE contains, in addition to these, architecture-specific constraints that check for the semantic consistency of the final design. These constraints are represented independently in a Constraint Checker subsystem (Fig. 2). For example, the dependency between customization parameters mentioned earlier is specified as the following constraint in KASE:

Parameter(m-Create-Layout, layout-algorithm) = nested => Parameter(m-Create-Layout, node-shape) = rectangle

(If the *layout-algorithm* in module m-Create-Layout has been specified as nested, then the *node-shape* must be a rectangle.)

We have developed a representation scheme for representing and classifying these constraints in KASE. (For details see (Nakano, 1993)). At any stage in the design process, KASE can ask a user to check for the consistency of the design with respect to a set of constraints. KASE computes all the constraints that are violated, groups them under the various categories, and presents them to the user. The user then has to initiate design actions to remove the constraint violations.

6. Results

Once the customization process is completed and no constraint violations are detected by KASE, a designer can ask KASE to generate code for his problem. KASE uses the values of the customized parameters to instantiate the set of generic procedures in all the modules of the architecture. The set of all instantiated procedures constitutes the solution to the problem instance. For procedures that were customized manually the designer has to provide the relevant code.

We have used several examples to produce new kinds of diagramming tools by customizing the generic diagramming architecture. The following is an example of a problem specification which has been used to synthesize a new diagramming tool in KASE:

Problem specification: Knowledge-sources and Events diagram
Input:- x : τ_1
Parameters:-
τ_1 : module
O : {k | knowledge-source(k) & provides(x,k)} $\cup$
    {e | $\exists$ k, knowledge-source(k) & event(e) &
        event-posted(k,e) & provides(x,k)} $\cup$
    {e | $\exists$ k, knowledge-source(k) & event(e) &
        trigger(k,e) & provides(x,k)}
R1: {(k, e) | event-posted(k,e)} U {(e,k) | trigger(k,e)}
R2: {}
Parameter constraints: Irreflexive(R1) & Anti-symmetric(R1)

(For readability we have provided a declarative specification for the output of the lambda expression used to instantiate the parameters O and R1. In practice the user has to provide a lisp function that computes the relevant sets.)

The input to the above problem is a module *x*. The objects that need to be shown in the layout consist of all knowledge-sources provided by the module *x* as well as all events that are triggered or posted by the knowledge sources. Only one binary relation R1 needs to be shown. R1 is defined as follows: *k* R1 *e* if *e* is an event triggering knowledge source *k* or if *e* is an event posted by *k* .

During customization the user has to provide the following information:

1. The format of a menu item and conditions for its invocation (i.e. those modules on which a user clicks to get the menu item). This is used by m-Diagram-Manager to update the relevant menus.

2. The layout algorithm to be used. (The default algorithm suggested by KASE is the nearest-neighbor algorithm.)

3. The shapes of the graphical icon to be used for depicting the knowledge-sources and events. The user can specify additional properties for the icon, e.g. the size and whether it is to be highlighted or not.

4. The kind of arc needed to show the relation between knowledge sources and events (the default suggestion is a directed arrow since the relation is not symmetric.) In addition, specification of other properties like whether the arc should be labeled or not, the format of the label, etc. are also part of the customization.

5. The kinds of editing to be allowed on the objects (The user has to provide the code to modify the internal data structure of knowledge-sources and events while KASE automatically generates the code interfacing it to the m-Diagram-Manager.)

The visible effects of the execution is the appearance of a new menu item when the appropriate objects are clicked. Upon selecting the new item, KASE uses the newly generated code to draw a new kind of diagram.

We have been using the generic architecture for the GDP domain to synthesize several different diagrams at a significantly increased efficiency than before. Moreover, other project members besides the assigned programmer have been able to generate new kinds of diagrams. We believe that this exercise has been useful in showing that it is possible to reduce the maintenance burden on programmers by using the KASE approach.

7. Discussion and Related Work

There are several papers that describe systems that help users in designing user interfaces for an underlying application. Some of the systems use a rule-based approach like KASE to establish the mapping between application objects and displayed objects or *widgets* (e.g. (Bennet,Boies,Gould,Greene, & Wiecha, 1989)). These systems are aimed at general, application-independent techniques for building user interfaces. In our work, we were interested in generating graphical displays for a restricted application since the focus of our research is primarily in software reuse, and not in user interface technology.

KASE is based on a framework for developing software systems in which generic architectures are the fundamental unit of reuse. In this respect our work is related to the Domain-Specific Software Architecture (DSSA) project. However, as far as we know, KASE is the first system that has (1) demonstrated how the concepts of software architectures, domain models, and software synthesis can be integrated in a unified framework, and (2) shown the applicability of the framework in the design of two different systems based on generic architectures in two different application domains.

The idea of constructing software systems by first capturing a model of a class of systems was first presented in a system called Draco (Neighbors, 1984). KASE specializes the Draca approach by defining a domain in terms of a problem-class for a generic architecture.

KASE is also related to application generators which seek to automate the synthesis of components within a narrow application domain. Application generators may be thought of as high-level compilers for narrow-spectrum, application-specific languages. They are well suited for domains where a set of requirements can be easily expressed in some simple, high-level language. However, since the knowledge about the application domain is embedded in the macros and interpreters of the application generator and the compilation process is transparent to an end-user, it is difficult to adapt them for different application. In KASE, the knowledge for customizing an architecture is represented explicitly as rules and methods which makes it much more flexible than application generators.

On the other hand, it seems to indicate that the KASE approach is not appropriate for all problem classes. Its utility depends critically on the complexity of acquiring and representing the relevant customization knowledge and ensuring its correctness. At one extreme, there are classes of problems that are well understood and for which the customization knowledge would be relatively complete and correct. In such domains an application generator approach would be more efficient than KASE. On the other extreme are entirely new classes of problems for which little is known regarding the design process. In such domains, the customization knowledge would be very sparse and KASE could not offer much assistance beyond that offered by the current CASE technology. Thus, it seems that KASE would be most useful for domains that lie between these two extremes.

A closely related project to KASE is LEAP (Graves, 1991) which also uses architectures as a basis for synthesizing systems and relies on an interactive designer to synthesize a specific system. However, in LEAP there is no explicit representation of a model for a class of problems. Consequently, the customization knowledge in LEAP is not as rich as in KASE; on the other hand LEAP is able to learn the relevant rules dynamically during design.

8. Conclusions

We have presented an approach to software reuse that is based on abstracting the design of a class of problems as a generic architecture. Such an approach provides reuse at the level of entire systems in addition to reuse at the level of algorithms or subroutines.

We have demonstrated the practical utility of our approach by being able to successfully reuse the design knowledge as well as code for a set of problems in solving new problem instances in a realistic and useful domain. We are currently applying KASE to model another application concerned with the analysis of radio signals obtained from planetary missions. This application will enable us to get empirical evidence on the usability of KASE by users outside the KASE group.

Acknowledgment

This research was supported in part by a grant from NASA Ames Research Center. Nelleke Aiello was primarily responsible for implementing the customization knowledge for this domain. E. Parra, G. Nakano, and R. Guindon contributed in implementing the various layout diagrams.

References

Barstow, D. (1979). *Knowledge based program construction*. New York: Elsevier North Holland.

Bennet, W.,Boies, S.,Gould, J.,Greene, S., & Wiecha, C. (1989). Transformations on a Dialog Tree: Rule-Based Mapping of Content to Style. In *Proc. of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, (pp. 67-75). Williamsburg, VA:

Bhansali, S., & Nii, H. P. (1992a). KASE: An integrated environment for software design. In *2nd International Conference on Artificial Intelligence in Design*, Pittsburgh, PA:

Bhansali, S., & Nii, H. P. (1992b). Software Design by Reusing Architectures. In *7th Knowledge-Based Software Engineering Conference*. McLean, Virginia:

Graves, H. (1991). Lockheed Environment for Automatic Programming. In *6th Annual Knowledge-Based Software Engineering Conference*, (pp. 78-89). Syracuse, NY:

Lubars, M. D. (1991). The ROSE-2 Strategies for Supporting High-level Software Design Reuse. In M. R. Lowry & R. D. McCartney (Eds.), *Automating Software Design* AAAI Press/The MIT Press.

Mark, W.,Tyler, S.,McGuire, J., & Schlossberg, J. (1992). Commitment Based Software Development. *IEEE Trans. on Software Engineering*, 18(10), 870-885.

Mettala, E. (1990). Domain Specific Software Architectures Unpublished report.

Nakano, G. (1993). Consistency Maintenance mechanism in KASE. In *Sixth Annual Florida AI Research Symposium (FLAIRS-93)*. Ft. Lauderdale, Florida:

Neighbors, J. (1984). The DRACO approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(9), 564-573.

Shaw, M. (1990). Toward higher-level abstractions for software systems. *Data & Knowledge Engineering*, 5, 119-128.

Smith, D. R. (1990). KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1024-1043.